XR on Linux

Robust mutex implementation under Linux and scalability

Modification History

Revision	Date	Originator	Comments
0.1	02/01/10	Etienne Martineau	

Theory:

- A mutex is a 32 bit lock variable; Atomic operation manipulate the lock.
- The "lock/unlock" fast path doesn't involve kernel.
- The "lock" slow path involve calling into kernel sys_futex(FUTEX_WAIT). Internally it place the running thread on a wait queue;
- The "unlock" slow path involve calling into kernel sys_futex(FUTEX_WAKE). Internally, it parse the wait queue list, find the desired thread and place it on the run queue ;

Robust mutex implementation:

- A special attribute is required to initialized the mutex "robust"; (pthread_mutexattr_setrobust_np)
- User space (libc) maintain a per-thread private list of robust mutext object;
- The list is a simple linked list;
- At process init time, libc pass the list pointer to the kernel;
- At process termination time, kernel parse the list [doesn't trust it] and mark all contended mutex owned by the current thread with the OWNER_DIED flag.
- Threads that are block on the defunct object return from the pthread_mutex_lock with `EOWNERDEAD`

Scalability:

- When a process died:
    - On a recent x86 machine kernel takes 130msecs to parse 1 millions contended mutex and mark them all with OWNER_DIED
    - On a recent x86 machine kernel takes 30msecs to parse 1 millions uncontended mutex
- At run time:
    - Creating/removing mutex object involve walking a linked list.
    - No system call required
    - The list is malloc in user space
- The slow path for both lock/unlock involve kernel parsing the wait queue list; That list is hashed with the user space mutex address therefore it scale well.

Priority inheritance:

- Under Linux the default scheduling policies is sched_other. This is a fair scheduling algorithm (time-sharing scheduler). This type of scheduling policies cannot cause priority inversion.
- There is other type of scheduling policies like FIFO or RR part of the real-time categories. With that type of scheduling policies, it's possible to create priority inversion.
- The Kernel doesn't automatically protect against priority inversion but instead the application that register itself into the real-time scheduling classes must explicitly used "pthread_mutexattr_setprotocol (&attr, PTHREAD_PRIO_INHERIT);" in order to set the mutex object to inherit the priority.

Conclusion:

- Mutex subsystem of Linux scale well.
- For application using large amount of mutex object there is a performance hit but at least it is contained to the running application only. There is no global taxes.
- The only potential area where the mutex subsystem has scalability problem is at process termination time because kernel is subject to an intensive loop: Fortunately, unlike other approach (QNX), kernel is fully reentrant meaning that the "taxes" of parsing the list in kernel is almost ALL contained to the current user context. Moreover the list is RCU based meaning lock free for read only access.